BOOTLOADERS

ESE516: IoT Edge Computing

Monday March 25, 2019

Eduardo Garcia- egarcia@seas.upenn.edu

SLOW EUROPEAN CLOCKS

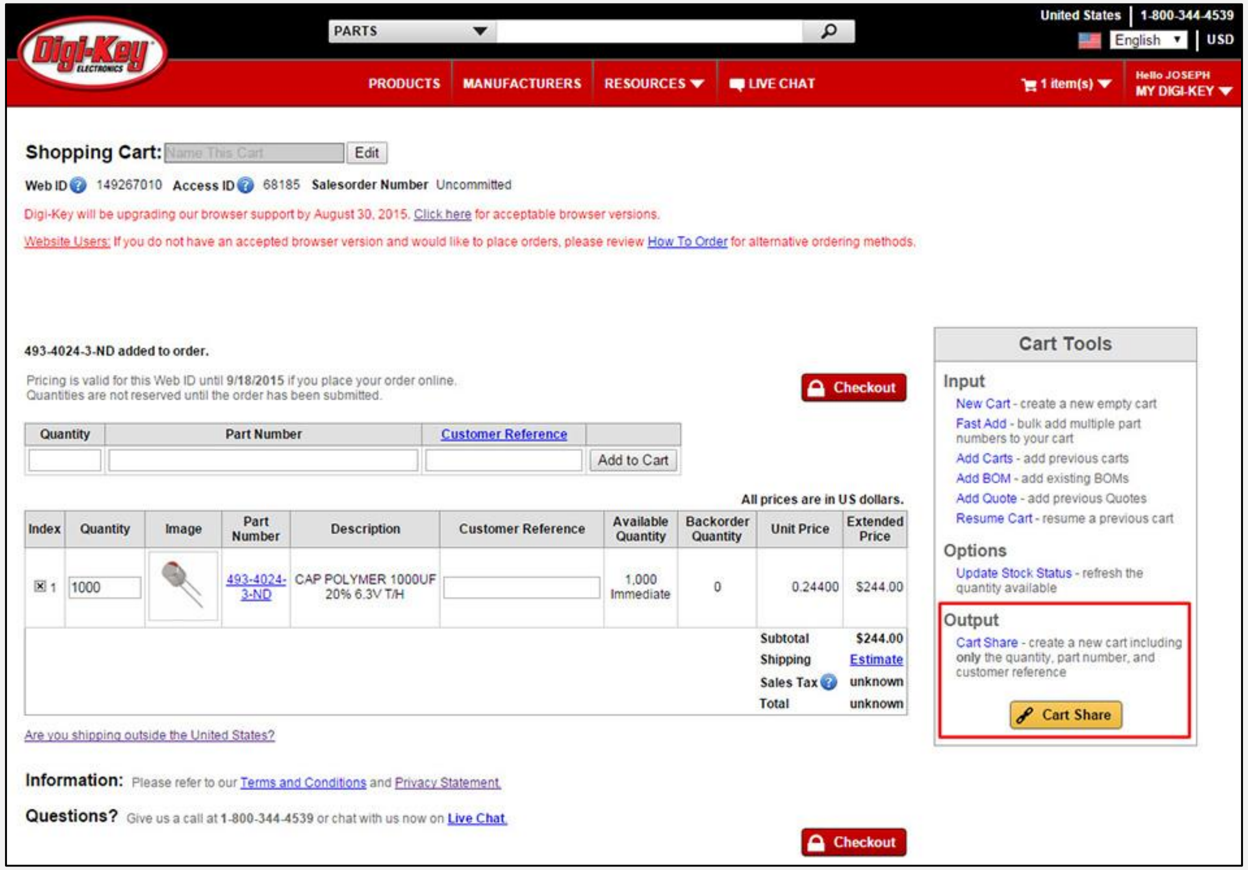
- Power is supplied at 60Hz (like USA) or 50Hz (Japan, Europe)
- Serbia should be handling the electric grid they share with Kosovo
- Serbia does not recognize Kosovo as independent, and they haven't been handling the additional power draw
- This has caused a dip in the supplied power frequency! 49.996Hz
- Appliances sync with the power grid, so a different supplied power frequency results in different time!
- 6 minutes off as of now…

https://www.nbcnews.com/news/world/time-slowing-down-europe-here-s-why-n855076

https://99percentinvisible.org/episode/you-should-do-a-story/

THROUGH HOLE COMPONENTS

- We need to order these for you, as PCB.ng is only SMT.
- Please add your list by End Of Day today!



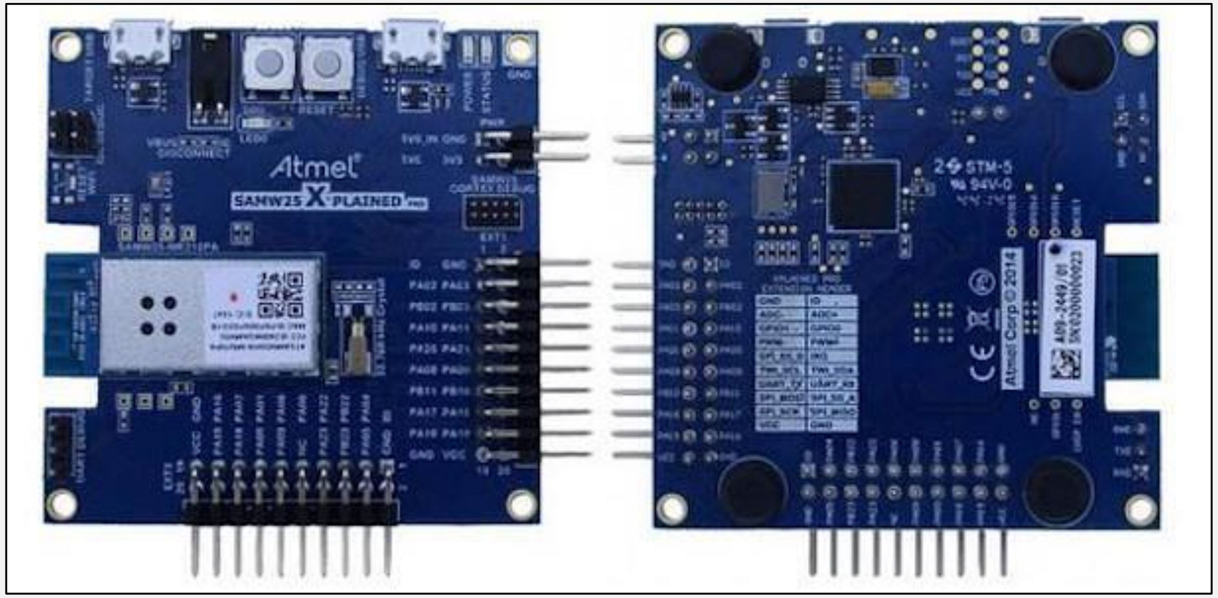
LAYING SOME GROUNDWORK

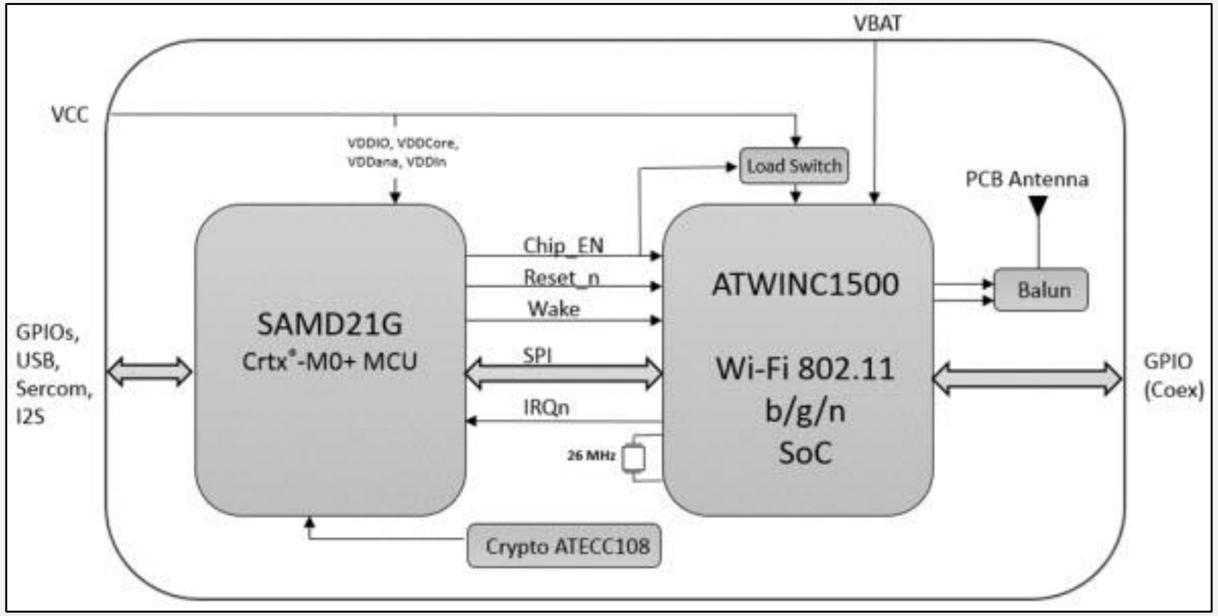
TERMS

- **Flash** or **External Flash**: the SPI Flash chip, where we'll be storing our firmware images
- **Non Volatile Memory** or **NVM**: the flash memory within the MCU, on the same silicon as the RAM
- **CRC**: Cyclic redundancy check, a data integrity method
- **WDT**: watchdog timer, a failsafe mechanism

SAM W25

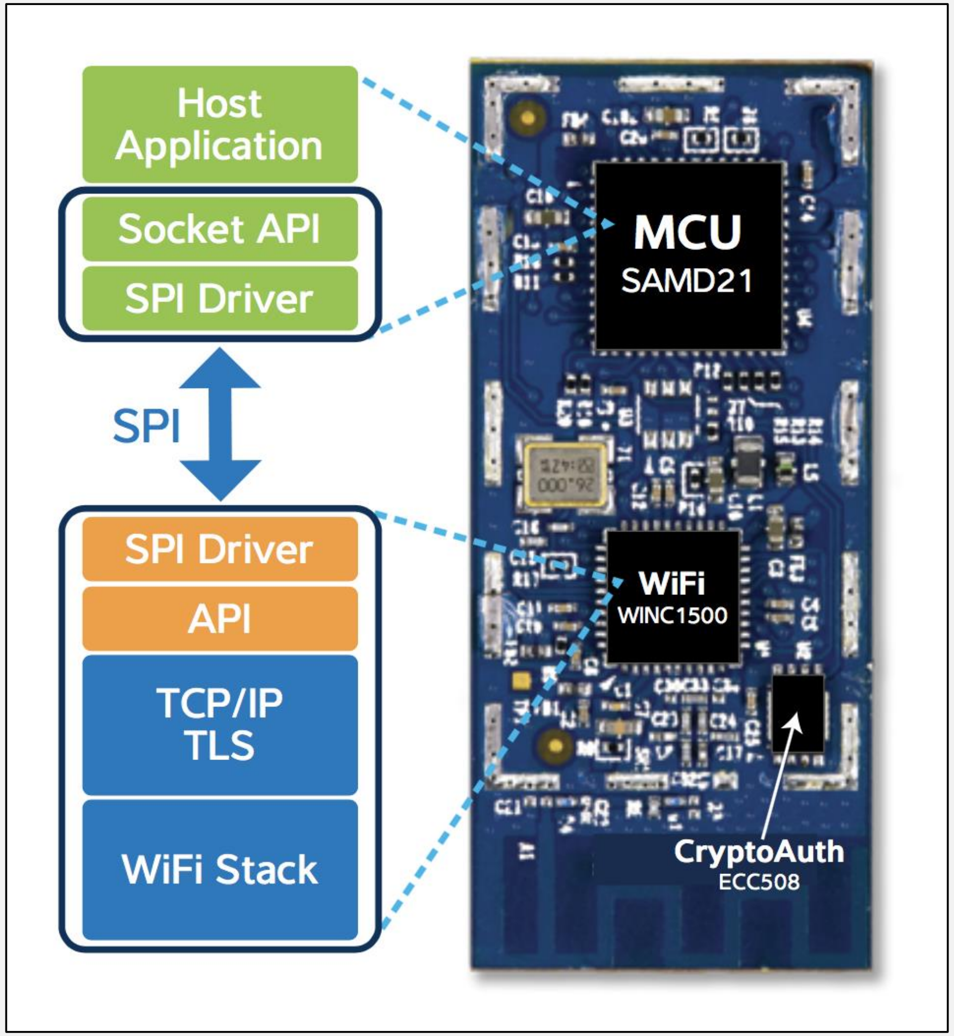
- It's a **module** that combines:
  - **SAMD21:** Cortex M0+
  - **ATWINC1500:** WiFi silicon from Atmel
  - **ATECC108:** Hardware crypto chip



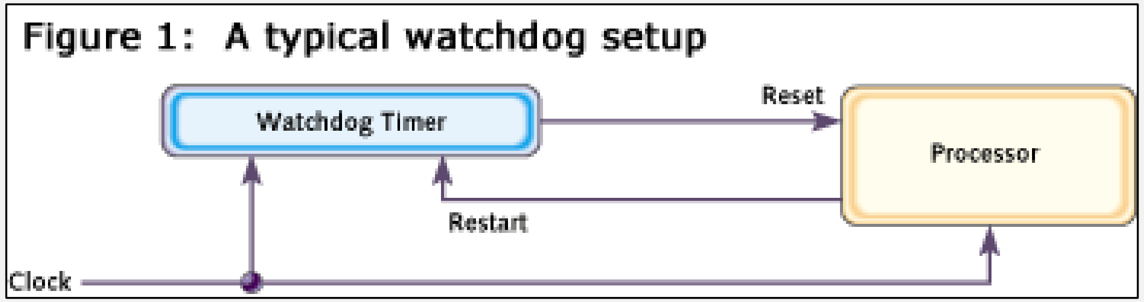


SAM W25

- SPI interface to the WINC1500
- WiFi stack is located on the WINC1500 – this is good because it doesn't take up application code space, RAM space
- Updating the WINC1500 firmware is a matter of shuttling the images to the MCU, then out of the MCU on SPI to the WINC1500



WATCHDOG TIMER



- Feed the dog; if the dog doesn't get fed, it'll get angry and reset the MCU.
- Good way of catching bad behavior
  - If you get caught in the weeds
  - Ex: Get cause in a while(1) loop or hard fault, the dog doesn't get fed
- Can be totally external to MCU (hardware), or (more typically) a high priority within the MCU firmware

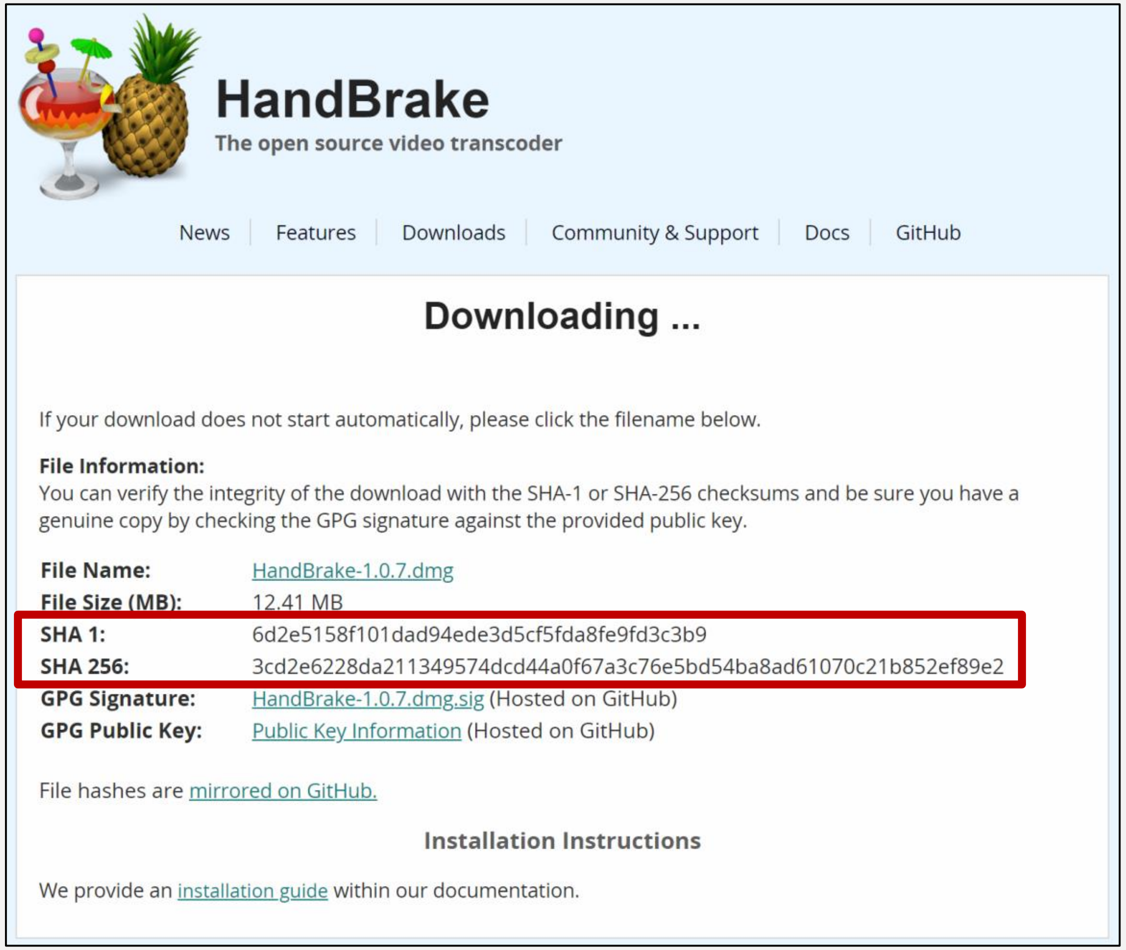
[...] sometimes called "kicking the dog."

The appropriate visual metaphor is that of a man being attacked by a vicious dog.

If he keeps kicking the dog, it can't ever bite him
But he must keep kicking the dog at regular intervals to avoid a bite.

CHECKSUM

- A way of checking file integrity after transmission
- Is the received file EXACTLY like the source?
- Has the file been tampered with? Or were bits dropped?
- Run a function across the entire data to generate a unique fingerprint of that data



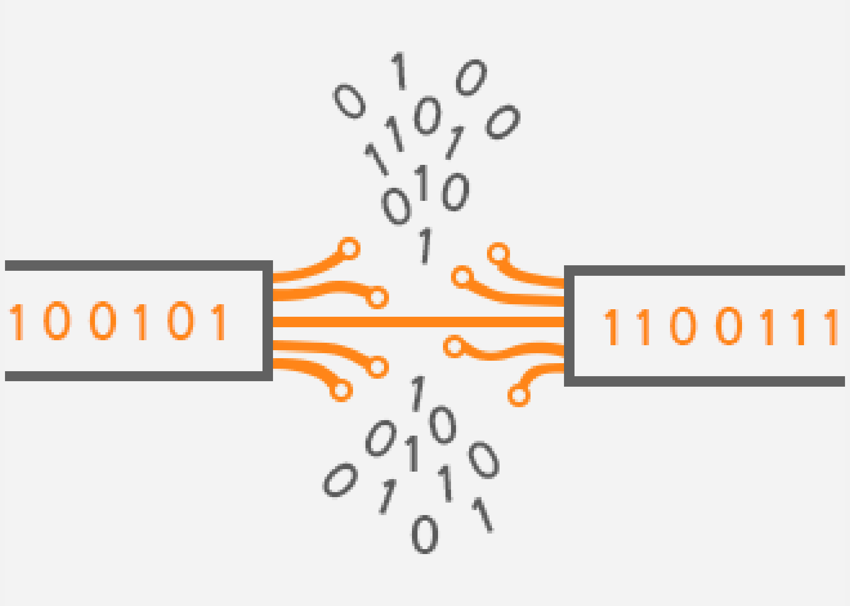
CHECKSUM

- CRC32: Cyclic Redundancy Check, 32-bit
  - Based on remainder of polynomial division
- You'll be using this throughout your code when you're reading and writing
  - Verifying firmware downloaded correctly
  - Verifying external flash wrote correctly
  - Verifying NVM wrote correctly
- Atmel Studio Framework has a module!

```c
enum status_code crc32_recalculate(const void *data, size_t length, crc32_t *crc)
{
  const word_t *word_ptr =
      (word_t *)((uintptr_t)data & WORD_ALIGNMENT_MASK);
  size_t temp_length;
  crc32_t temp_crc = COMPLEMENT_CRC(*crc);
  word_t word;

  // Calculate for initial bytes to get word-aligned
  if (length < WORD_SIZE) {
    temp_length = length;
  } else {
    temp_length = ~WORD_ALIGNMENT_MASK & (WORD_SIZE - (uintptr_t)data);
  }

  if (temp_length) {
    length -= temp_length;

    word = *(word_ptr++);
    word >>= 8 * (WORD_SIZE - temp_length);
    temp_crc = _crc32_recalculate_bytes_helper(word, temp_crc, temp_length);
  }

  // Calculate for whole words, if any
  temp_length = length & WORD_ALIGNMENT_MASK;

  if (temp_length) {
    length -= temp_length;
    temp_length /= WORD_SIZE;

    while (temp_length--) {
      word = *(word_ptr++);
      temp_crc = _crc32_recalculate_bytes_helper(word, temp_crc, WORD_SIZE);
    }
  }

  // Calculate for tailing bytes
  if (length) {
    word = *word_ptr;
```

SHA1 COLLISION

- Happened in February 2017
- SHA1 is used to track authenticity
- Google was able to create two PDFs with different content, but the same SHA1 fingerprint
- Imagine if your legal documents could be modified without your knowledge?
- https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html



PC & SP & RV

PROGRAM COUNTER (PC)

- Holds the address for the next instruction to execute
- Incremented for each execution

STACK POINTER (SP)

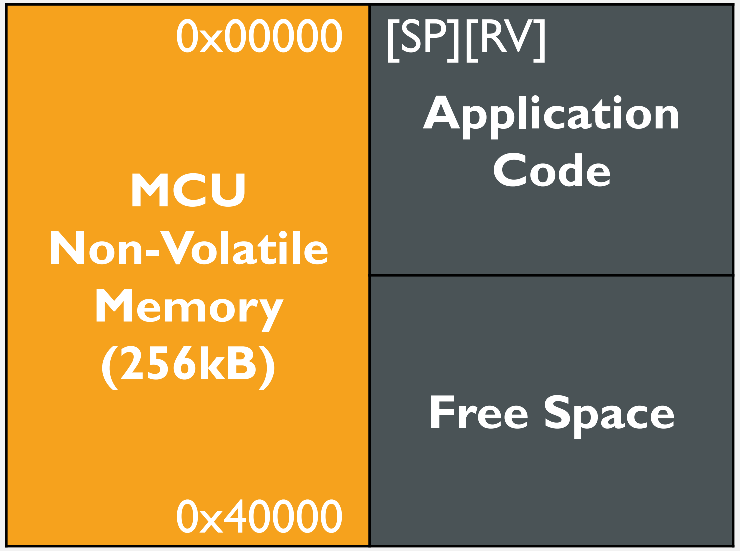
- Memory pointer
- Stacks store data top down

RESET VECTOR (RV)

- Handles code execution on device reset
- Default **word** location where the MCU will look

HOW DOES OUR MCU BOOT?

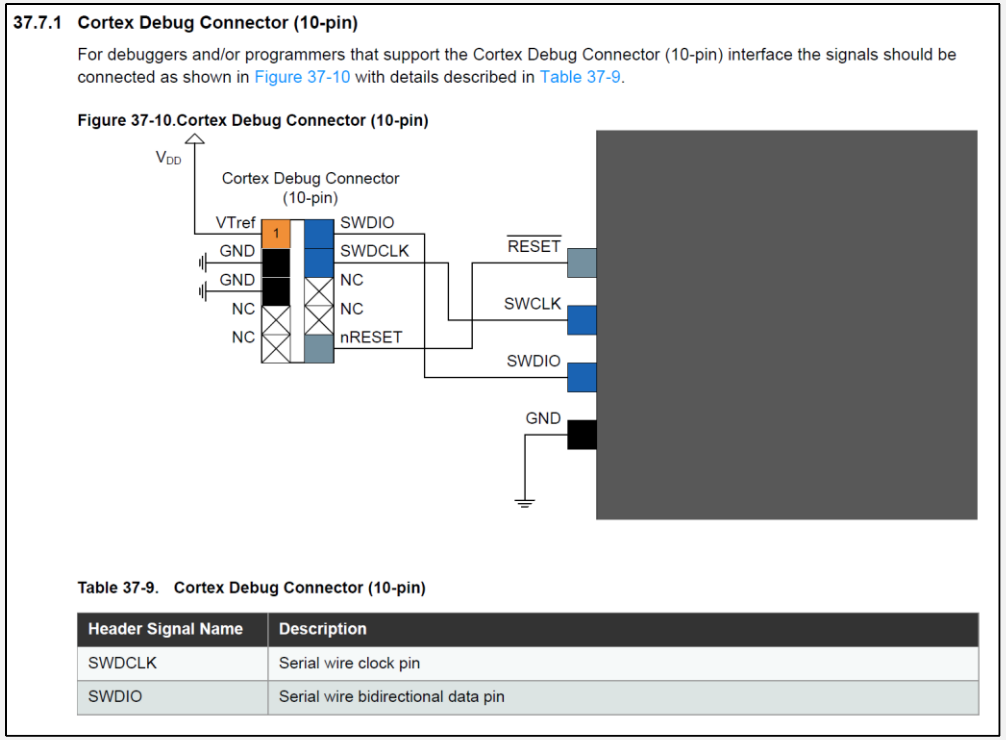
1. After power up, the device is held in reset until power is stabilized.
   1. Then, 1MHz clock from internal 8MHz RC oscillator divided by 8
2. I/O pins are tri-stated (high Z) after power up
3. After reset is released, CPU fetches Program Counter (PC) & Stack Pointer (SP) from reset address – **0x00000000**
   1. **First executable address in internal flash.**



BOOTLOADERS

LOADING CODE

- We need to get the internal memory set up with the correct bits.
- We use a debugger to write this internal memory.
- **Serial Wire Debug** is the two pin protocol we're using.
- **JTAG** is another common programming protocol.



37.7.1 Cortex Debug Connector (10-pin)

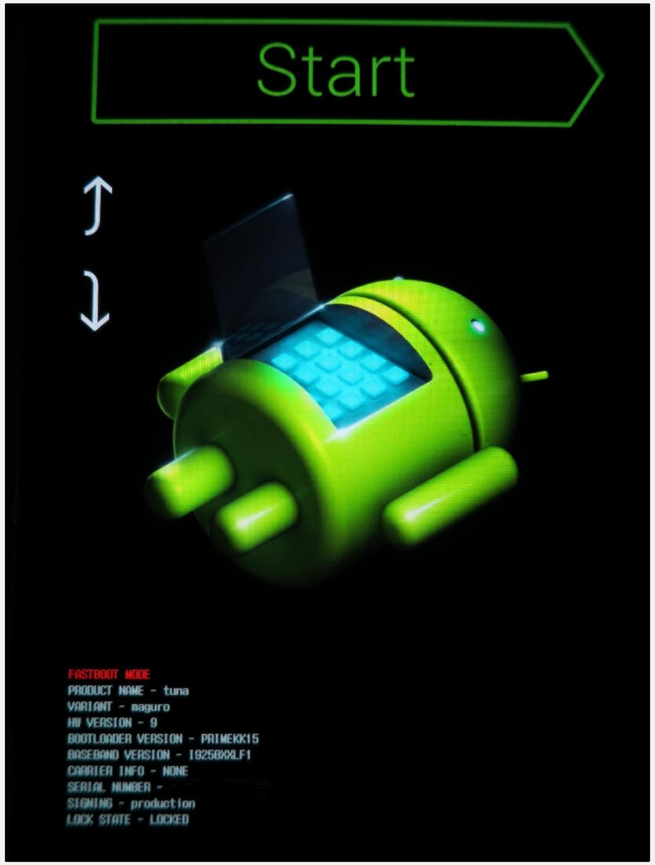
For debuggers and/or programmers that support the Cortex Debug Connector (10-pin) interface the signals should be connected as shown in Figure 37-10 with details described in Table 37-9.

Figure 37-10.Cortex Debug Connector (10-pin)

Table 37-9. Cortex Debug Connector (10-pin)

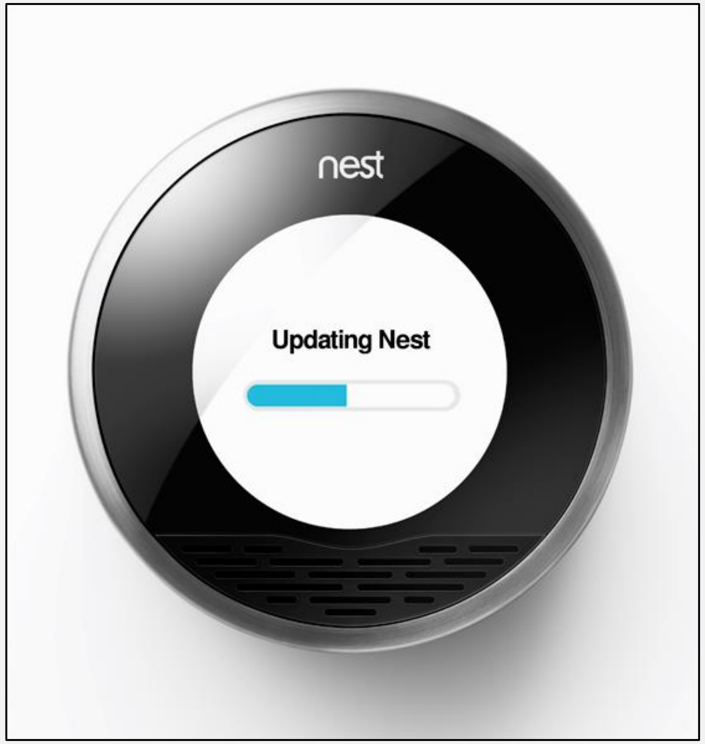
Header Signal Name	Description
SWDCLK	Serial wire clock pin
SWDIO	Serial wire bidirectional data pin

ARDUINO™


Start
FASTBOOT MODE
PRODUCT NAME - tuna
VARIANT - maguro
HW VERSION - 9
BOOTLOADER VERSION - PRIMEKK15
BASEBAND VERSION - I9250XXLF1
CARRIER INFO - NONE
SERIAL NUMBER -
SIGNING - production
LOCK STATE - LOCKED

WHAT IS A BOOTLOADER?

- Special bit of code that runs before the application code runs
- Can handle writing new application code to the microcontroller
- Circumvents need for sometimes expensive & unavailable debuggers
  - USB, WiFi, Cellular, etc.
- User accessible way of updating devices
  - Not everyone is an EE =)



WHY IS IT IMPORTANT?

- Electronics are now living devices -- they are not limited to the firmware they're manufactured and assembled with
- Features can be enabled after the fact (or with an extra payment *Oscilloscopes*!)
- Bugs & security flaws can be patched

WHERE ARE BOOTLOADERS USED?

- Home gadgets: Nest, TVs
- Fitbits, Pebble, smart watches
- Tesla! And cars in general..
- Cell phones (cellular network)

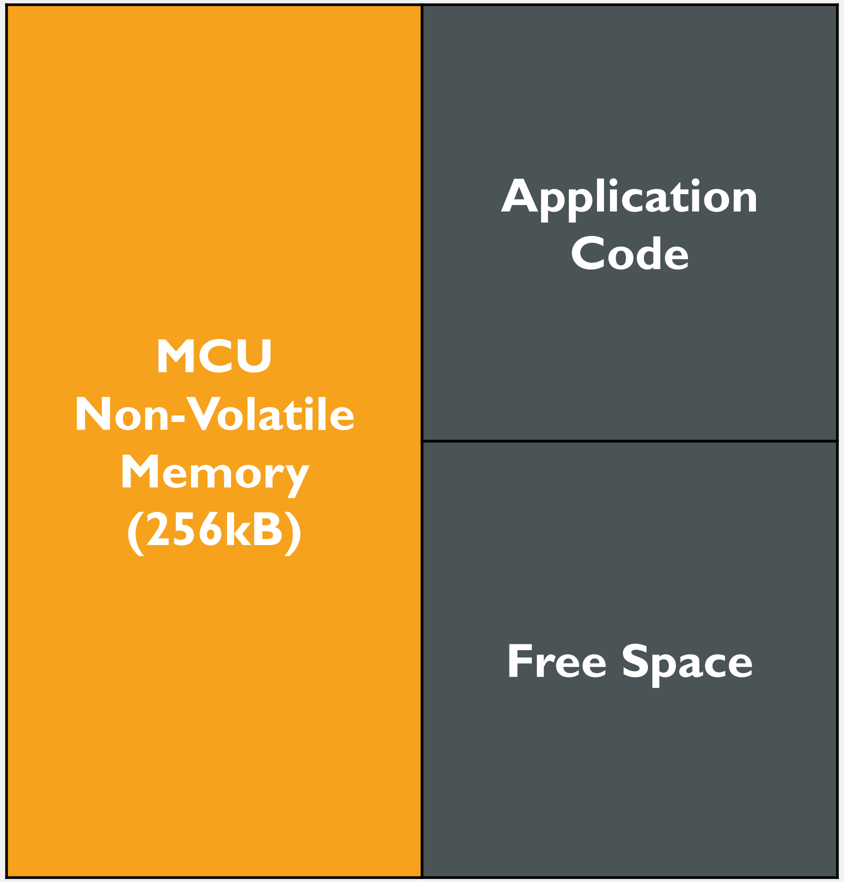
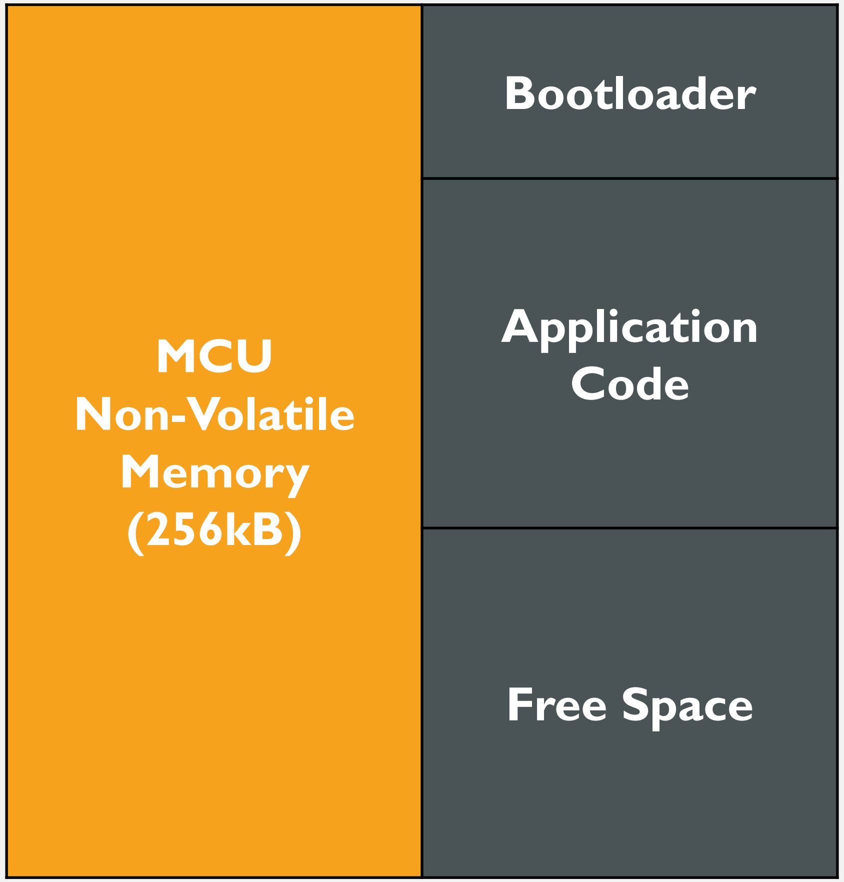
WHITEBOARD TIME

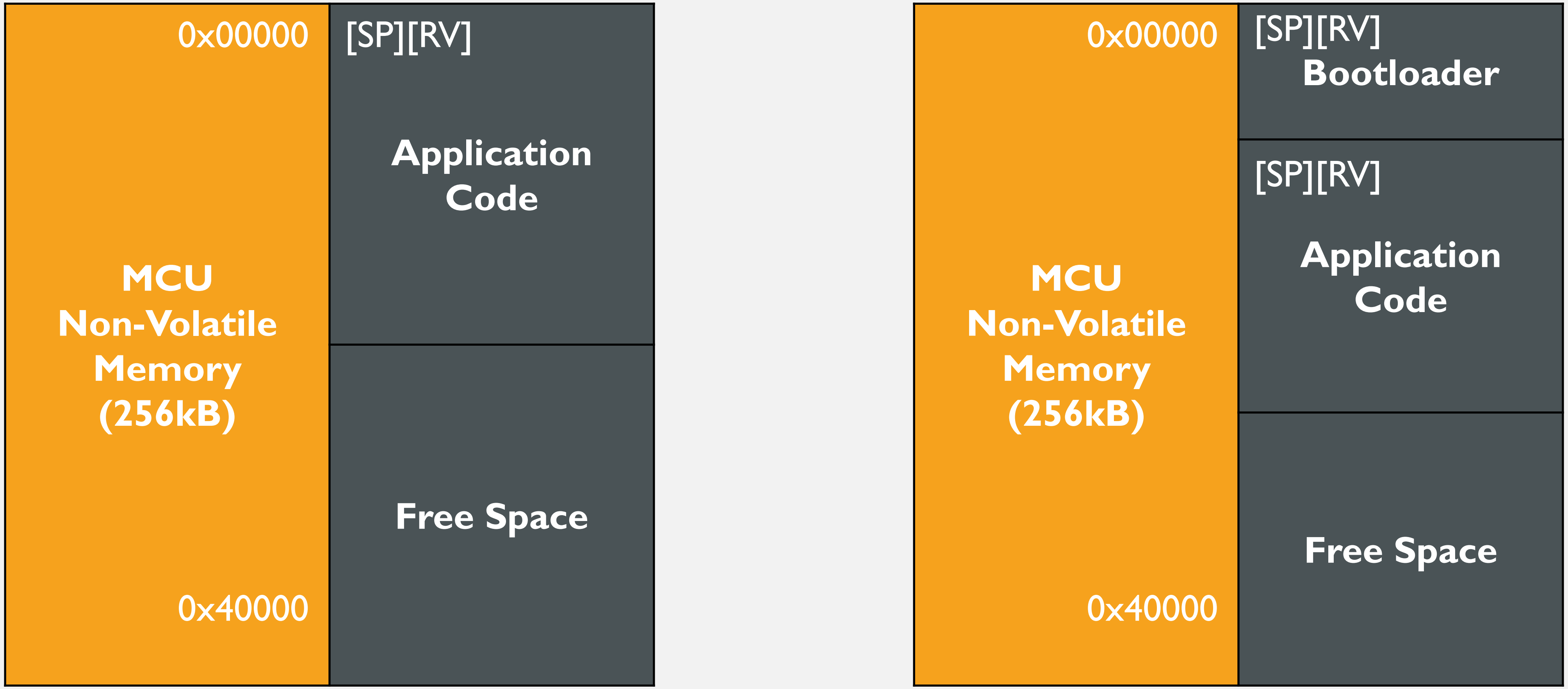
WHERE IS THE BOOTLOADER LOCATED?

HOW DO WE DOWNLOAD THE FIRMWARE?

WHERE DO WE STORE THE DOWNLOADED FIRMWARE?

HOW DO WE WRITE THE FIRMWARE TO MEMORY?


MCU
Non-Volatile
Memory
(256kB)
Application
Code
Free Space

MCU
Non-Volatile
Memory
(256kB)
Bootloader
Application
Code
Free Space


0x00000
[SP][RV]
Application Code
MCU Non-Volatile Memory (256kB)
Free Space
0x40000
0x00000
[SP][RV]
Bootloader
[SP][RV]
Application Code
MCU Non-Volatile Memory (256kB)
Free Space
0x40000

HOW DOES OUR MCU BOOT?

- First, check the 0x0000 – the initial address in memory.
- In this example, the application code lives at 0x2000. To run this code, we must "rebase" the stack pointer to point to the new start of code.

Figure 1-1. Memory Map of ATSAMD21J18 with an Application and SAM-BA with both USB and UART

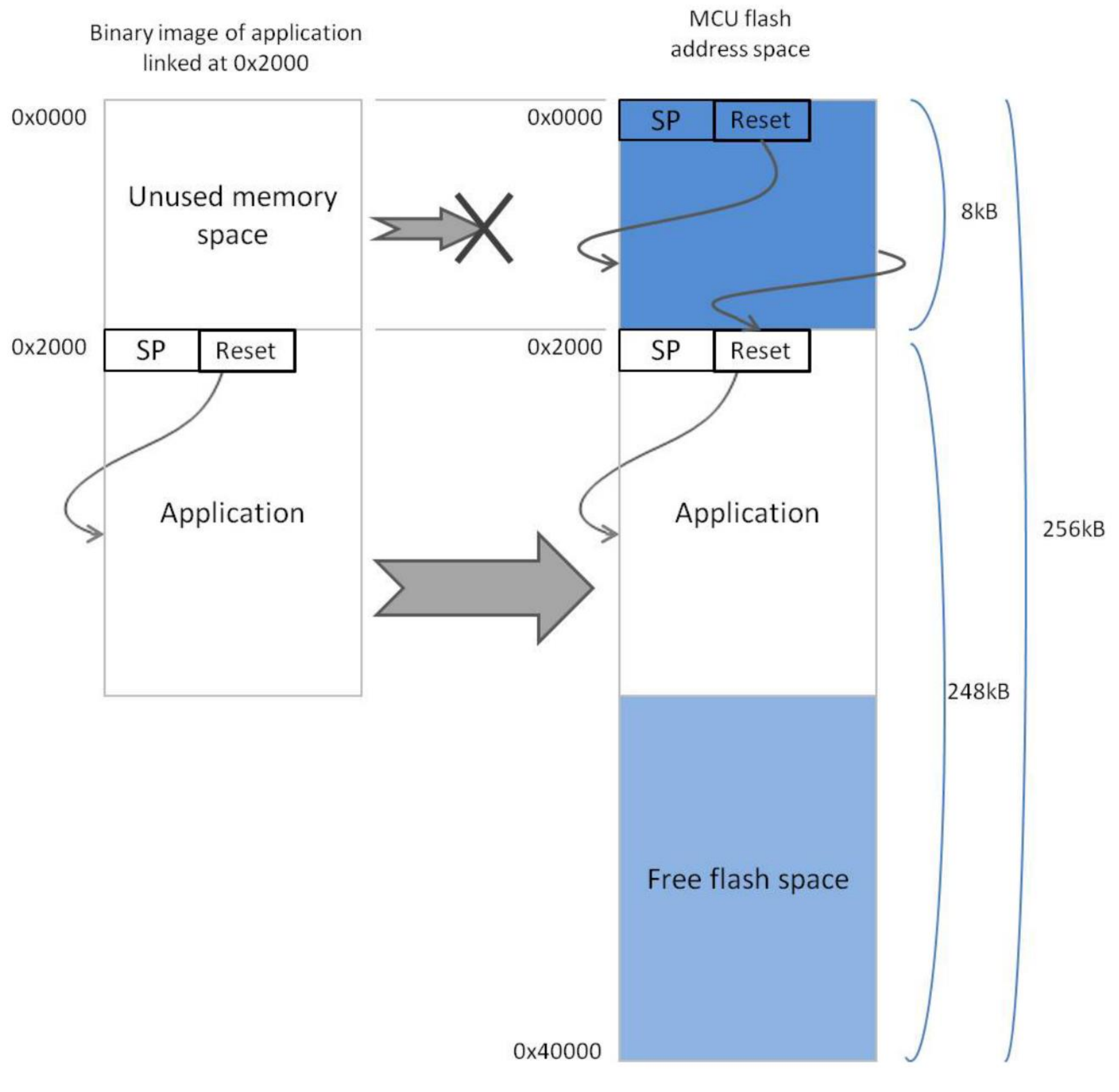
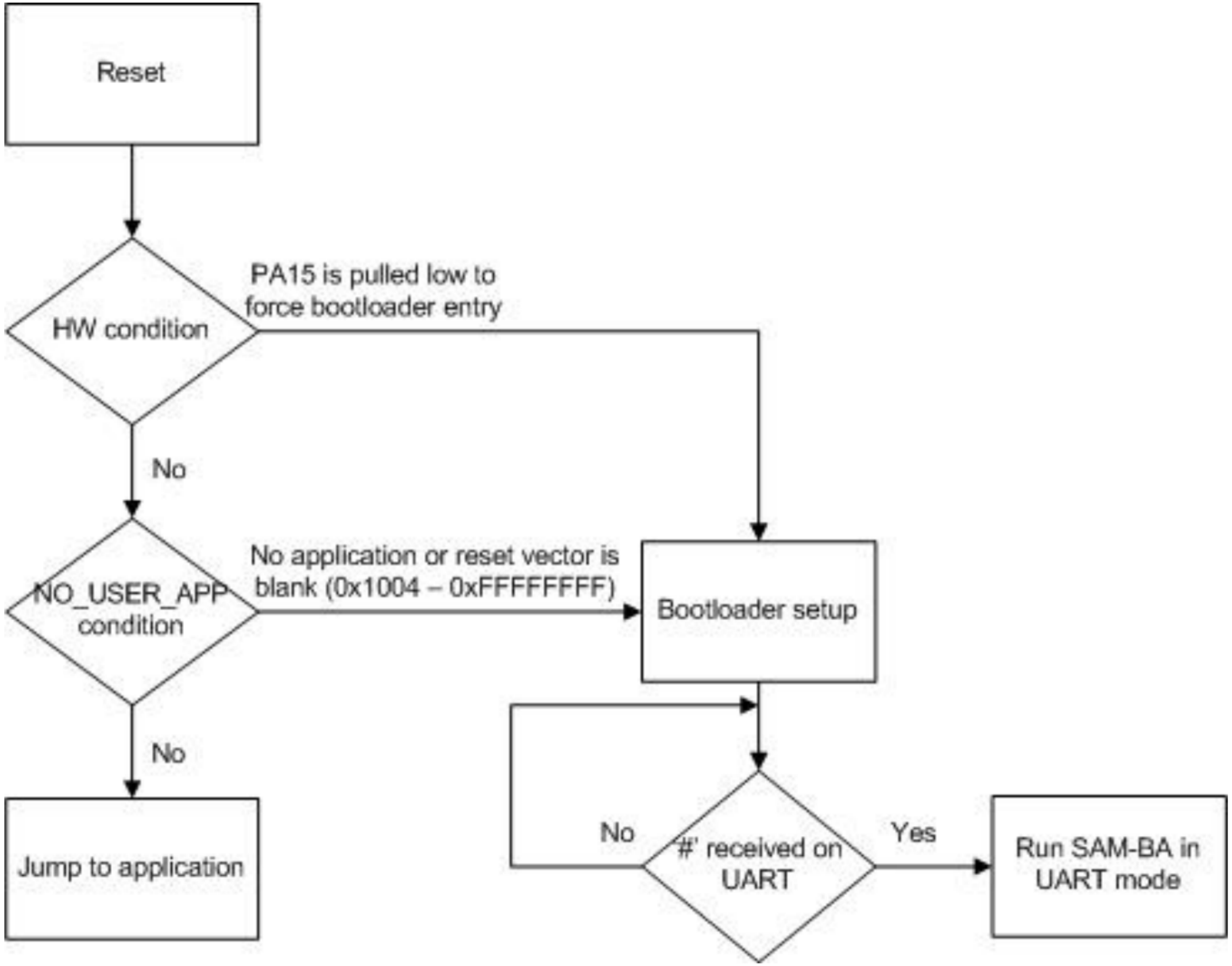


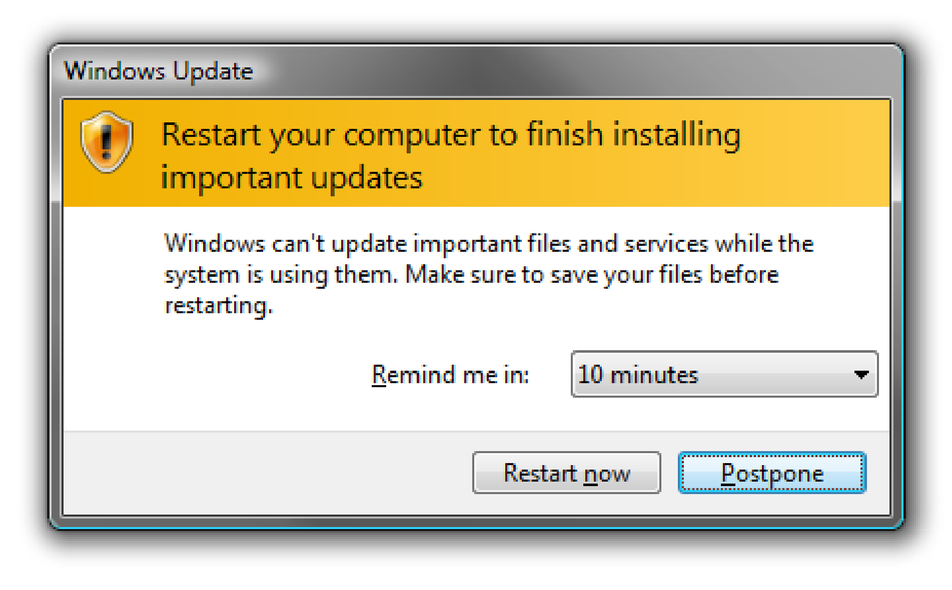
Figure 2-4. Boot Process of Atmel SAM-BA using UART



<u>Code security concerns:</u> When SAM-BA monitor is entered, it allows read and write access to the entire memory map of the device. It also allows the host to upload and execute software (applets) on the device.

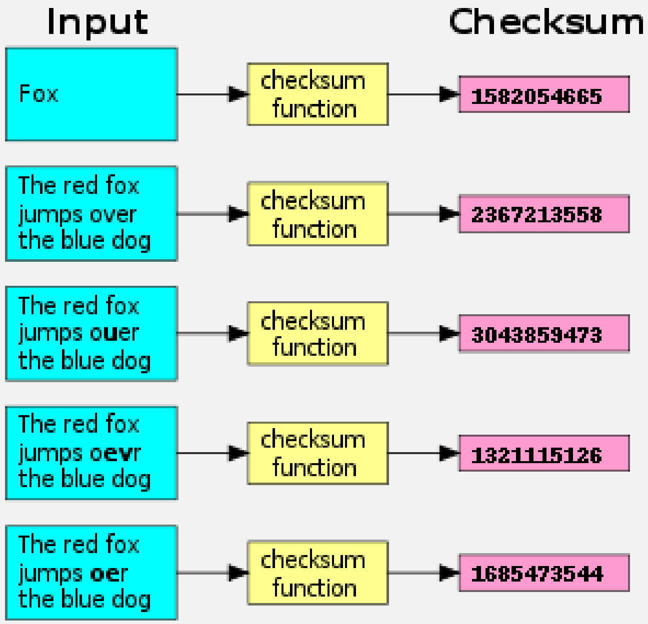
WHEN WILL YOU CHECK FOR UPDATES?

- Scheduled updates
  - Ex: Every night at 2am, just like Windows
- Button press / hold
- Command Line Interface
- Any sensor input



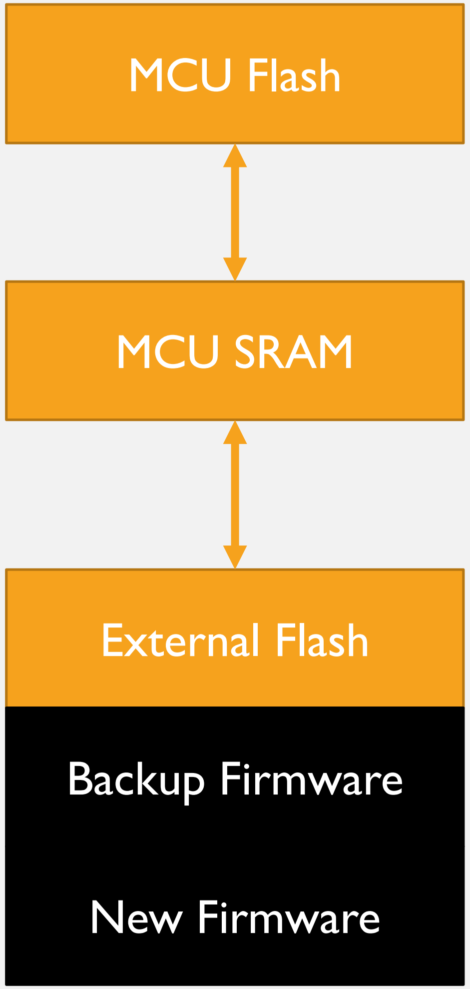
WHAT HAPPENS IF THE DOWNLOAD GETS CORRUPTED?

- Perhaps the data connection is poor, or the device loses power in the middle of a download.
  - You might get part of the file, or corrupted bits!
- Verify with a checksum
  - Iterate over all the data and crunch it into one number that can be compared with a base.
  - Reject if the numbers don't match.
- We'll be using a CRC32 checksum
  - Useful not only for firmware download, but also sensor data upload and actuator data download
  - Lightweight, good for embedded



WHAT HAPPENS IF THE FIRMWARE IMAGE IS BAD?

- You should have a backup plan in the event that the new firmware image somehow breaks the build.
  - Track the reason the device reset.
  - You can track the number of watchdog timer resets – if the device resets 5, 10 times in a row due to WDT, it might be a sign that the firmware is bad
- If you can programmatically determine a "bad" firmware, you can load the backup, verified firmware
  - You can have a "golden image" that never gets overwritten and will always get the device into a functioning state
  - Track the previous best image.
- Expose USB or UART physically to the user and allow the bootloader to download a new firmware image.



DETAILS

BOOTLOADER PROCESS - PSEUDOCODE

int main(void)

Read the boot status flag

if (**boot_flag == UPDATE_FIRMWARE**){ Stay In Bootloader }

else if (**applicationCode is not loaded**){ Stay In Bootloader }

else if (**Bootloader Button is low**){ Stay In Bootloader }

else if (**battery level is not too low**) {Stay In Bootloader }

Firmware Image Checksum in External Flash

Update the application firmware

Firmware Image Checksum in NVM

BOOTLOADER RULES

1. Keep your bootloader as simple as possible.
2. Implement bootloaders into your project early – so you can test them thoroughly before shipping.

BOOTLOADER SIZE

- Bootloaders, ideally, should be small and fit into a tiny compiled footprint
- Atmel Studio 7 generated code is inherently large
  - Abstraction layers for ease of programming
  - Uses GCC, not as efficient as IAR or Kiel compilers
- You don't want to include everything in your bootloader – that'll bloat your binary
- Check the program memory usage in the Output panel after compiling
- Assume 0x2000 bytes for your bootloader

Output

Show output from: Build

```
    "C:\Program Files (x86)\Atmel\Studio\7.0\toolchain\arm\arm-gnu-toolchain\bin\arm-none-eabi-objcopy.exe" -O srec -R .eeprom -R
    "C:\Program Files (x86)\Atmel\Studio\7.0\toolchain\arm\arm-gnu-toolchain\bin\arm-none-eabi-size.exe" "Bootloader.elf"
       text    data     bss     dec     hex filename
       7644       8    6808   14460    387c Bootloader.elf
Done executing task "RunCompilerTask".
Using "RunOutputFileVerifyTask" task from assembly "C:\Program Files (x86)\Atmel\Studio\7.0\Extensions\Application\AvrGCC.dll".
Task "RunOutputFileVerifyTask"
            Program Memory Usage    :   7652 bytes   2.9 % Full
            Data Memory Usage       :   6816 bytes   20.8 % Full
```

COMPILING OPTIMIZATION

- Optimization is a neat way to reduce compiled code size & increase speed
  - Dead Code Elimination
  - Local and Global Common Subexpression Elimination
  - Constant Propagation
  - Partial Redundancy Elimination
  - Loop based optimizations like Loop Tiling, Loop Unrolling
- However, optimization can make it hard to debug – you may not be able to debug the exact line in question!
  - Create
- Learn more:
  - https://www.quora.com/Why-is-compiler-optimization-important
  - https://en.wikipedia.org/wiki/Optimizing_compiler

PARTITION TABLES

- Remember the caveats of your flash memory!
  - How small of a chunk of memory can you erase?
  - How many bytes can you write at a time?
  - You must erase before writing to a section of memory.
- What do you want to store in your status page?

SAMD21 256kB	0x00000	Bootloader
	0x01F00	Boot status
	0x02000	Application Code
	0x40000	End of memory

External Flash 8Mb	0x000000	Status Page
	0x001000	FW Slot #1 Header Page
	0x002000	FW Slot #1 Data (0x3E000)
	0x040000	FW Slot #2 Header Page
	0x041000	FW Slot #2 Data (0x3E000)
	0x7D0000	End of memory

RUNNING APPLICATION CODE

- On boot, the MCU will look to the 0x0000 address – this gives the stack pointer and reset handler for the bootloader
- To transition to the application code, we must point to the application code space – "rebase" the stack pointer & reset vector
- The code on the right handles this functionality
- USB MSC: Page 31-32 has some good references
  - http://www.atmel.com/images/atmel-42352-sam-d21-xpro-usb-host-msc-bootloader_training-manual_an8185.pdf

```c
/* Pointer to the Application Section */
void (*application_code_entry)(void);

/* Rebase the Stack Pointer */
__set_MSP(*(uint32_t *) APP_START_ADDRESS);

/* Rebase the vector table base address */
SCB->VTOR = ((uint32_t) APP_START_ADDRESS & SCB_VTOR_TBLOFF_Msk);

/* Load the Reset Handler address of the application */
application_code_entry = (void (*)(void))(unsigned *)(*(unsigned *)
(APP_START_ADDRESS + 4));

/* Jump to user Reset Handler in the application */
application_code_entry();
```

STATUS STRUCTURE

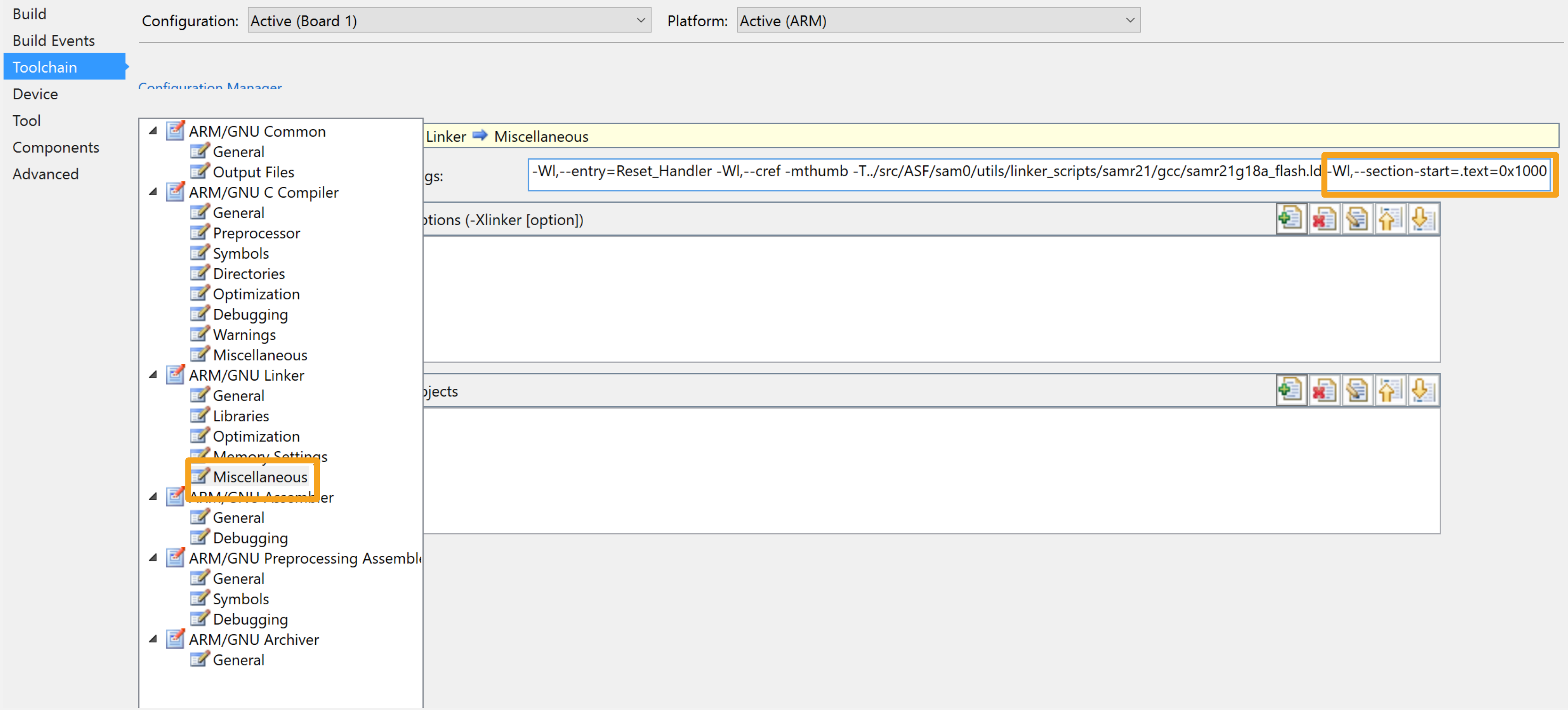
- Having a status / header page in memory is a nice way to organize metadata for the firmware binary
- For example, you can include your CRC32 for the entire binary, as well as the size of the file.
- FW_status handles what the current executing image is, the downloaded image, and whether or not I should be writing a new image.
- FW_header keeps track of versioning information – both firmware and hardware. It also holds the CRC for the associated FW image

```
typedef struct FW_status {
  uint8_t signature[3];      /// Used to determine that partition was initialized
  uint8_t executingImage;    /// Image 1 or 2 in the flash memory
  uint8_t downloadedImage;   /// Image 1 or 2 in the flash memory
  bool writeNewImage;        /// Is a new image ready to be written?
} FW_Status_T;


typedef struct FW_header {
  uint16_t firmwareVersion;
  uint16_t hardwareVersion;
  uint16_t checksum;
} FW_Header_T;
```

```
/// Read in the boot status
error_code = nvm_read_buffer(BOOT_STATUS_ADDRESS, NVM_buffer_read, NVMCTRL_PAGE_SIZE);
if(error_code != STATUS_OK)
{
  while(1);
}
memcpy(&bootStatus, NVM_buffer_read, sizeof bootStatus);
```

```
/// If boot status signature is incorrect, write to the first slot in FW
if( bootStatus.signature[0] != 0xAB
||  bootStatus.signature[1] != 0xAC
||  bootStatus.signature[2] != 0xAB)
{
  bootStatus.executingImage   = 2;
}

/// Determine the address to write to in flash
if(bootStatus.executingImage == 1)
{
  flashImageAddress           = FW_IMAGE2_DATA_ADDR;
  bootStatus.downloadedImage  = 2;
}
```

GENERATING A BOOTLOADER FIRMWARE IMAGE



BOOTLOADER IMPLEMENTATION STEPS

Our path to bootloading success

BOOTLOADER IMPLEMENTATION STEPS

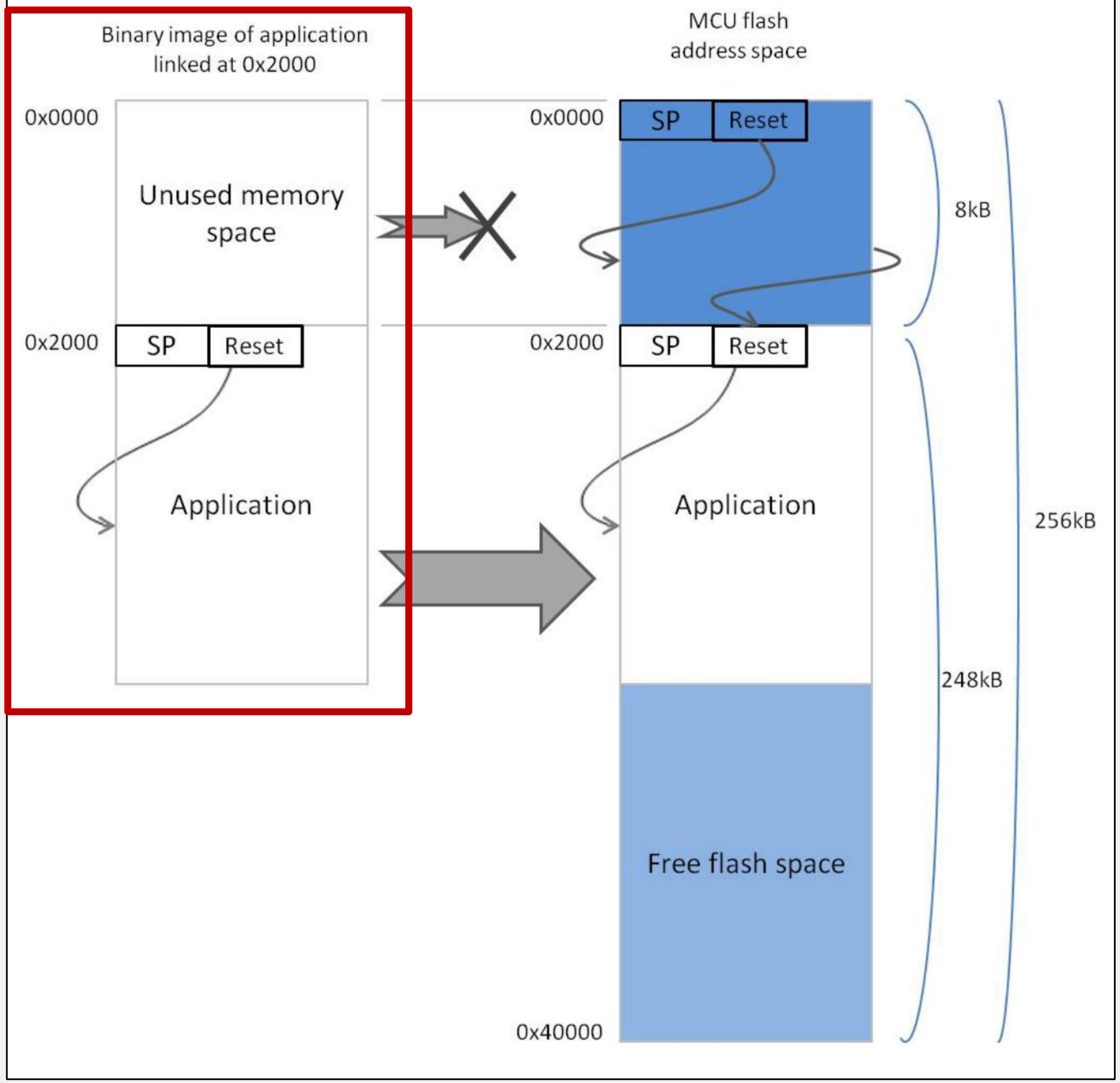
1. Develop firmware to jump from bootloader to application code
2. Develop firmware to read & write from external SD Card
3. Develop firmware to read & write from internal non-volatile memory (NVM)

1. BOOTLOADER TO APPLICATION

1. BOOTLOADER TO APPLICATION

- Create two projects within one solution for Atmel Studio
  - Bootloader
  - Application code
- Set up the linker for the start of the application code in memory – give yourself 0x2000 bytes of space for your bootloader

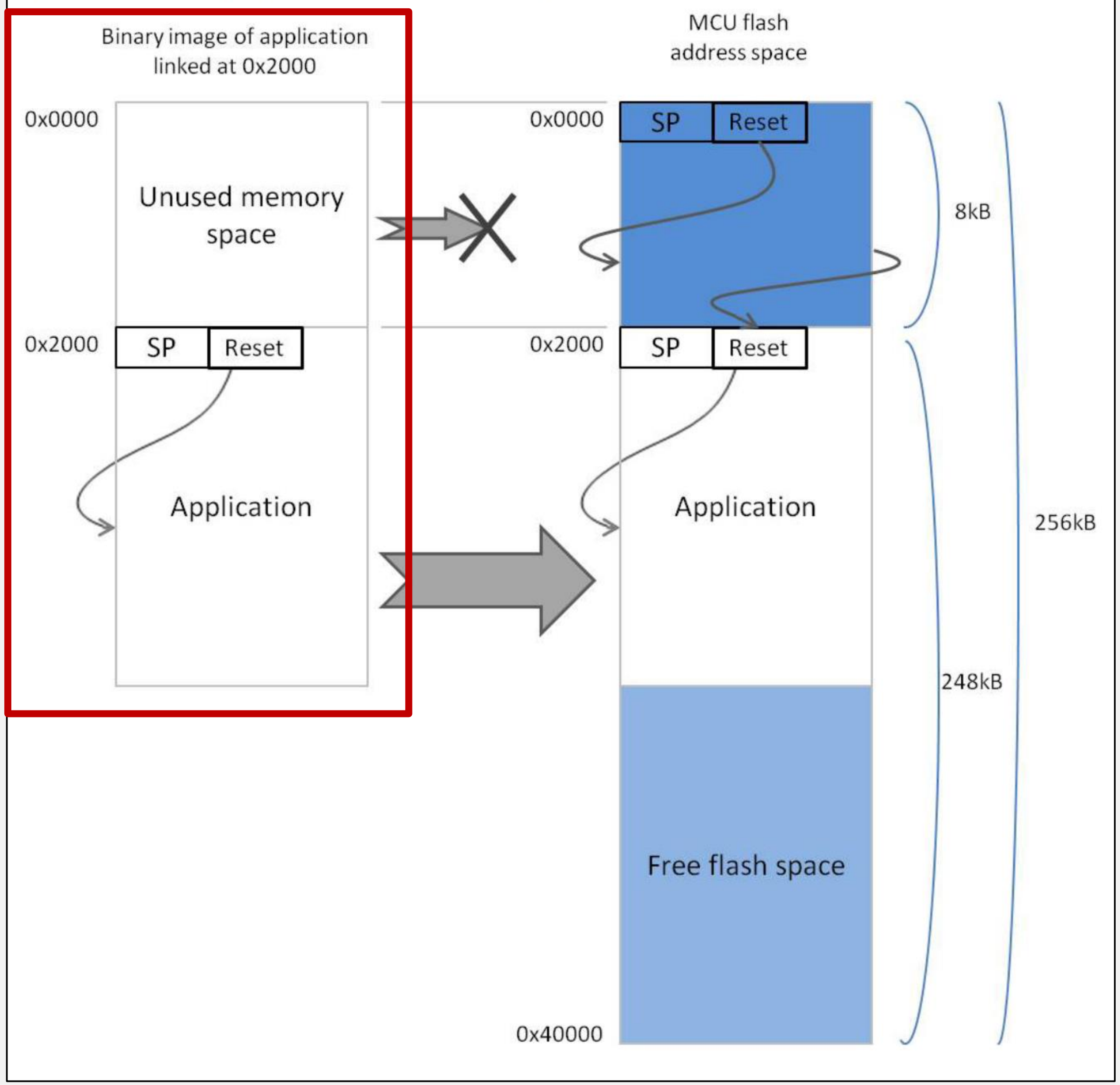
Figure 1-1. Memory Map of ATSAMD21J18 with an Application and SAM-BA with both USB and UART



1. BOOTLOADER TO APPLICATION

- Set up the bootloader to fully erase the memory
- Set up the application code to only erase the used memory

Figure 1-1. Memory Map of ATSAMD21J18 with an Application and SAM-BA with both USB and UART



2. EXTERNAL FLASH

2. EXTERNAL FLASH

- New this year – FatFS on an SD Card.
- FatFS is a FAT filesystem module. It abstract the difficulty of dealing with raw memory and having todo partition tables, and allows us to use files and folders, such as you would do on a "PC" program.

- I will post an example (starter code) of a FATFS system + SD Card stack for you to start with.

- If you need more information on FatFS, please see: http://elm-chan.org/fsw/ff/doc/appnote.html

3. NONVOLATILE MEMORY

2. NVM

- Set up the NVM communication module to write to internal memory
- Write a few pages worth of generated data
- Read the data back and verify that it's valid using a CRC32 calculation

FLASH MEMORY (IF WE USED IT)

FLASH MEMORY


adesto™
TECHNOLOGIES

8-Mbit
2.7V Minimum
SPI Serial Flash
Memory

AT25DF081A

- Flash memory access may be different than how you understand memory access
- We'll be using the adesto 8Mbit / 1MB flash IC for this discussion
  - If you used a different IC, YMMV
  - Many of these hex op codes (operational codes) are cross IC
- http://datasheet.octopart.com/AT25DF081A-SSH-T-Adesto-Technologies-datasheet-31984101.pdf

MEMORY ARCHITECTURE

The flash memory is chunked into different sectors.

Figure 4-1. Memory Architecture Diagram

Block Erase Detail

Internal Sectoring for Sector Protection Function	64KB Block Erase (D8h Command)	32KB Block Erase (52h Command)	4KB Block Erase (20h Command)	Block Address Range
64KB (Sector 15)	64KB	32KB	4KB	0FFFFFh – 0FF000h
			4KB	0FEFFFh – 0FE000h
			4KB	0FDFFFh – 0FD000h
			4KB	0FCFFFh – 0FC000h
			4KB	0FBFFFh – 0FB000h
			4KB	0FAFFFh – 0FA000h
			4KB	0F9FFFh – 0F9000h
			4KB	0F8FFFh – 0F8000h
		32KB	4KB	0F7FFFh – 0F7000h
			4KB	0F6FFFh – 0F6000h
			4KB	0F5FFFh – 0F5000h
			4KB	0F4FFFh – 0F4000h
			4KB	0F3FFFh – 0F3000h
			4KB	0F2FFFh – 0F2000h
			4KB	0F1FFFh – 0F1000h
			4KB	0F0FFFh – 0F0000h
64KB (Sector 14)	64KB	32KB	4KB	0EFFFFh – 0EF000h
			4KB	0EEFFFh – 0EE000h
			4KB	0EDFFFh – 0ED000h
			4KB	0ECFFFh – 0EC000h
			4KB	0EBFFFh – 0EB000h
			4KB	0EAFFFh – 0EA000h
			4KB	0E9FFFh – 0E9000h
			4KB	0E8FFFh – 0E8000h
		32KB	4KB	0E7FFFh – 0E7000h
			4KB	0E6FFFh – 0E6000h
			4KB	0E5FFFh – 0E5000h
			4KB	0E4FFFh – 0E4000h
			4KB	0E3FFFh – 0E3000h
			4KB	0E2FFFh – 0E2000h
			4KB	0E1FFFh – 0E1000h
			4KB	0E0FFFh – 0E0000h
⋮	⋮	⋮	⋮	
64KB (Sector 0)	64KB	32KB	4KB	00FFFFh – 00F000h
			4KB	00EFFFh – 00E000h
			4KB	00DFFFh – 00D000h
			4KB	00CFFFh – 00C000h
			4KB	00BFFFh – 00B000h
			4KB	00AFFFh – 00A000h
			4KB	009FFFh – 009000h
			4KB	008FFFh – 008000h
		32KB	4KB	007FFFh – 007000h
			4KB	006FFFh – 006000h
			4KB	005FFFh – 005000h
			4KB	004FFFh – 004000h
			4KB	003FFFh – 003000h
			4KB	002FFFh – 002000h
			4KB	001FFFh – 001000h
			4KB	000FFFh – 000000h

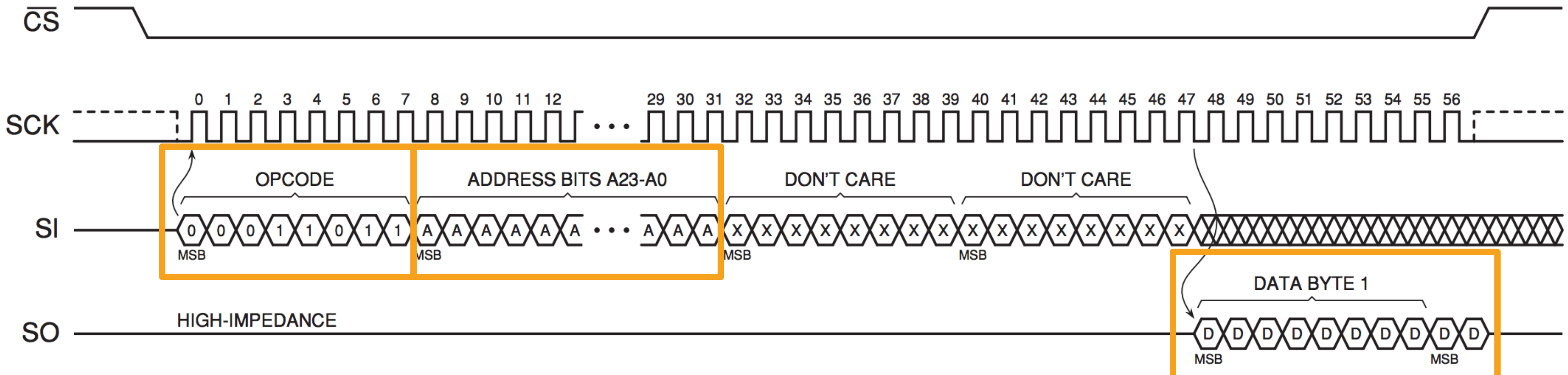
Page Program Detail

1-256 Byte Page Program (02h Command)	Page Address Range
256 Bytes	0FFFFFh – 0FFF00h
256 Bytes	0FFEFFh – 0FFE00h
256 Bytes	0FFDFFh – 0FFD00h
256 Bytes	0FFCFFh – 0FFC00h
256 Bytes	0FFBFFh – 0FFB00h
256 Bytes	0FFAFFh – 0FFA00h
256 Bytes	0FF9FFh – 0FF900h
256 Bytes	0FF8FFh – 0FF800h
256 Bytes	0FF7FFh – 0FF700h
256 Bytes	0FF6FFh – 0FF600h
256 Bytes	0FF5FFh – 0FF500h
256 Bytes	0FF4FFh – 0FF400h
256 Bytes	0FF3FFh – 0FF300h
256 Bytes	0FF2FFh – 0FF200h
256 Bytes	0FF1FFh – 0FF100h
256 Bytes	0FF0FFh – 0FF000h
256 Bytes	0FEFFFh – 0FEF00h
256 Bytes	0FEEFFh – 0FEE00h
256 Bytes	0FEDFFh – 0FED00h
256 Bytes	0FECFFh – 0FEC00h
256 Bytes	0FEBFFh – 0FEB00h
256 Bytes	0FEAFFh – 0FEA00h
256 Bytes	0FE9FFh – 0FE900h
256 Bytes	0FE8FFh – 0FE800h
⋮	
256 Bytes	0017FFh – 001700h
256 Bytes	0016FFh – 001600h
256 Bytes	0015FFh – 001500h
256 Bytes	0014FFh – 001400h
256 Bytes	0013FFh – 001300h
256 Bytes	0012FFh – 001200h
256 Bytes	0011FFh – 001100h
256 Bytes	0010FFh – 001000h
256 Bytes	000FFFh – 000F00h
256 Bytes	000EFFh – 000E00h
256 Bytes	000DFFh – 000D00h
256 Bytes	000CFFh – 000C00h
256 Bytes	000BFFh – 000B00h
256 Bytes	000AFFh – 000A00h
256 Bytes	0009FFh – 000900h
256 Bytes	0008FFh – 000800h
256 Bytes	0007FFh – 000700h
256 Bytes	0006FFh – 000600h
256 Bytes	0005FFh – 000500h
256 Bytes	0004FFh – 000400h
256 Bytes	0003FFh – 000300h
256 Bytes	0002FFh – 000200h
256 Bytes	0001FFh – 000100h
256 Bytes	0000FFh – 000000h

FLASH READ

- You can sequentially read out the entire memory
- Automatically incrementing address on every clock cycle
- Feed in an 8-bit op code, a 24-bit address -- get the data at that address

Figure 7-1. Read Array – 1Bh Opcode



FLASH READ

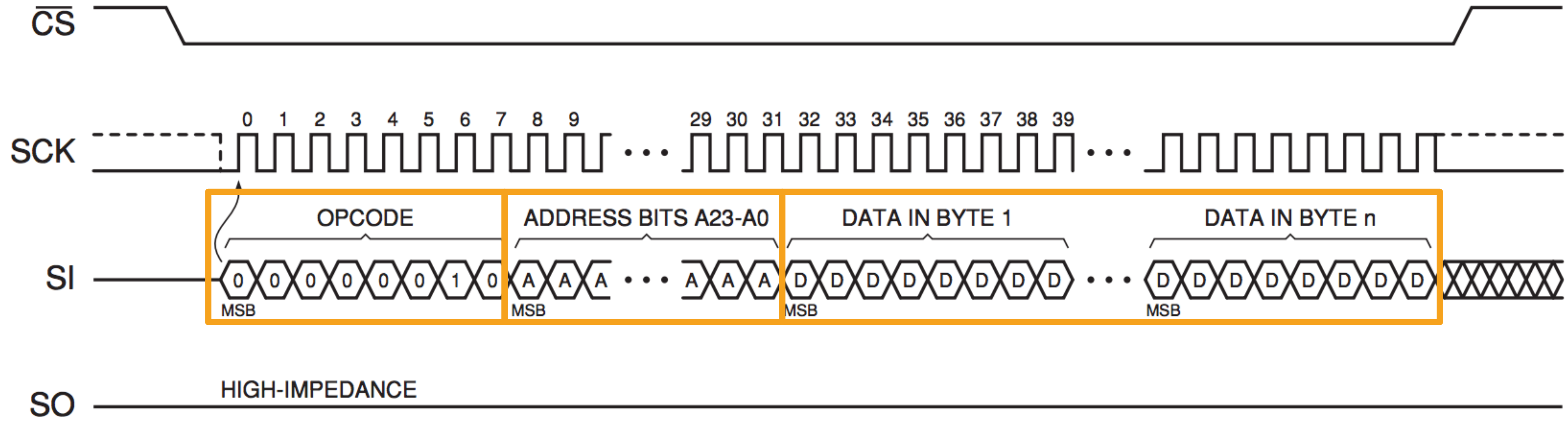
```c
void flash_id(void)
{
  /// Prep buffers
  writeBuffer[0] = DEVICE_ID_CMD;

  /// SPI Callback
  transfer_complete_spi_master = false;
  spi_select_slave(&spi_master_instance, &slave, true);
  spi_transceive_buffer_job(&spi_master_instance, writeBuffer, readBuffer, 4);
  while(!transfer_complete_spi_master);
  spi_select_slave(&spi_master_instance, &slave, false);
```

- Example of using SPI callback functions from Atmel Studio 7
- 4 bytes total
  - 1 byte for the op code
  - 3 bytes for the ID shifted out from the SPI flash

FLASH WRITE

- Can write up to 256-bytes at a time – or just 1 byte
- Write latch enable before
- Read the datasheet! Also, existing open source code.

Figure 8-2. Page Program



FLASH ERASE FUNCTIONS

- The smallest erasable block for this flash IC is 4 kB.
- So, if you want to edit even 1 byte in a block of data, you must erase the 4kB block before doing so.

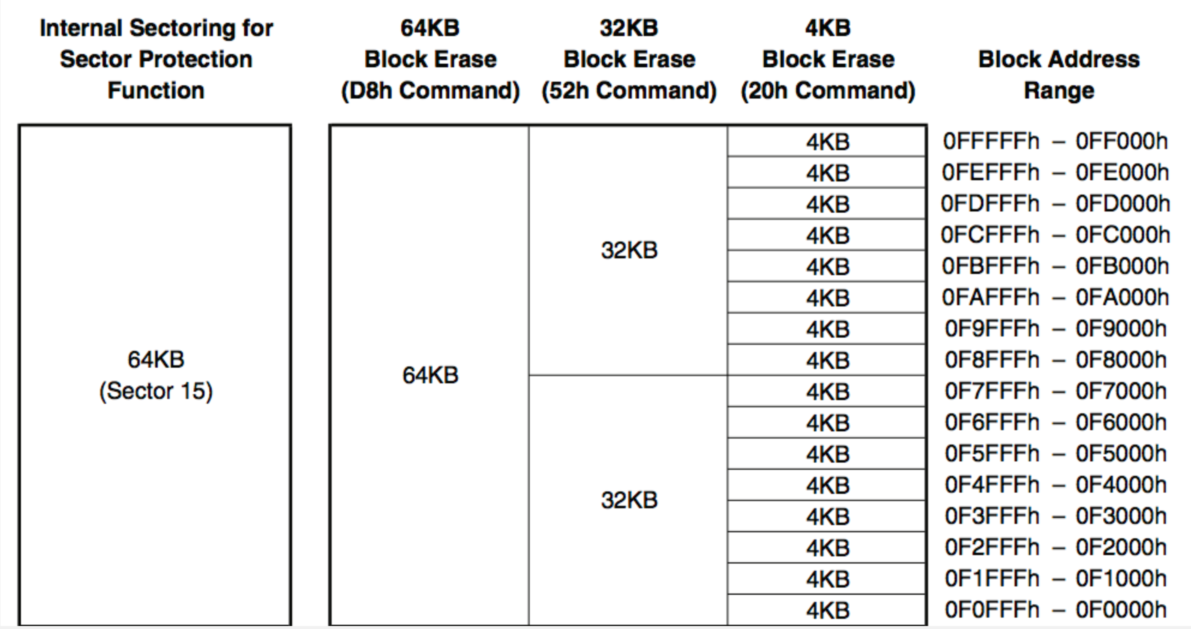
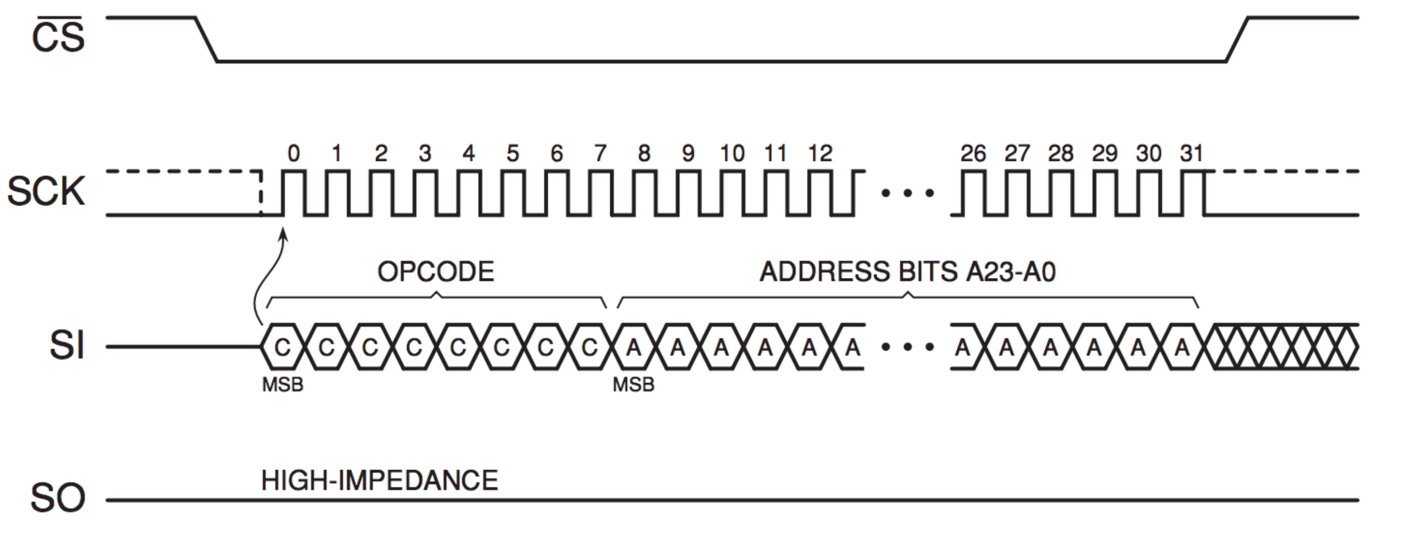


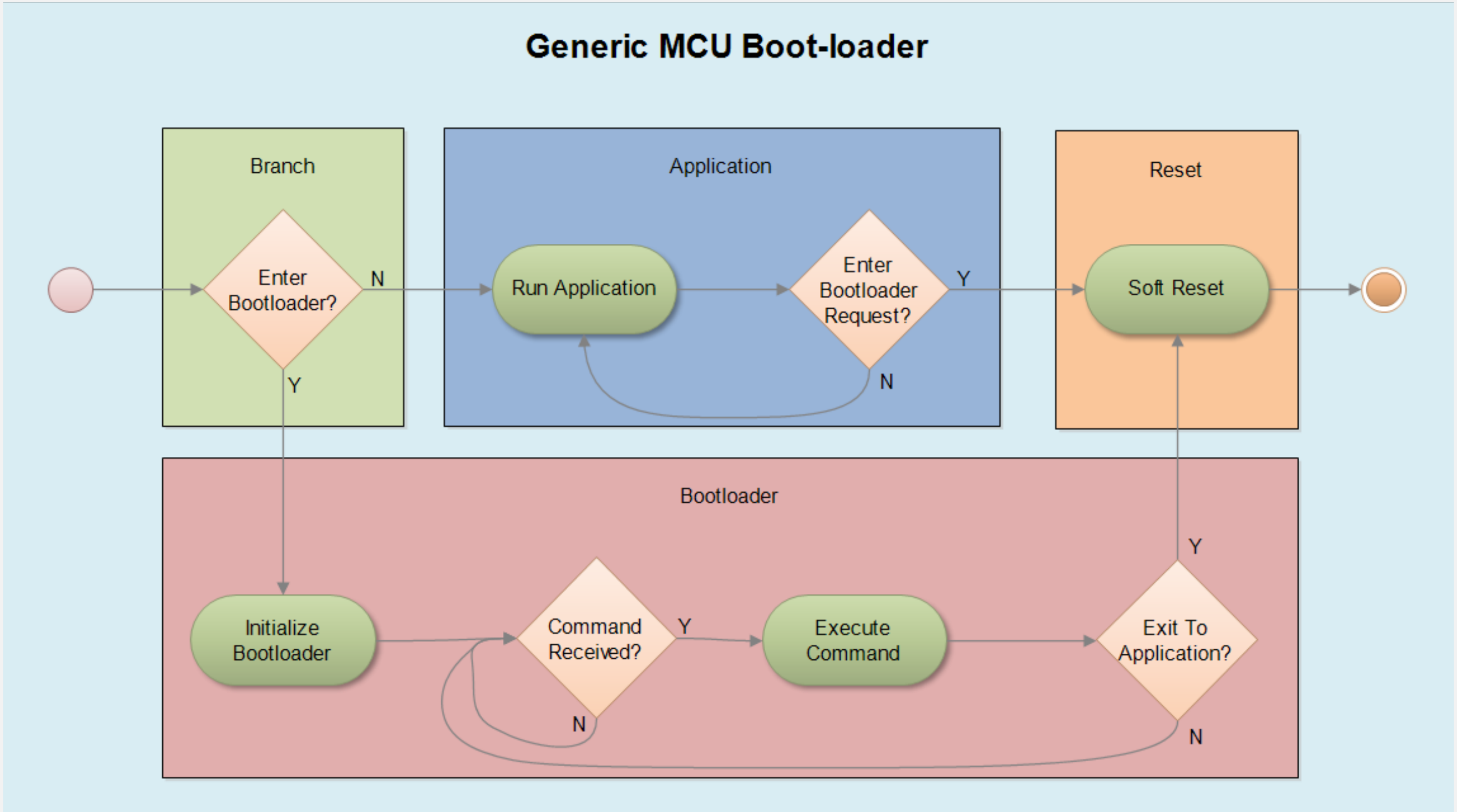
Figure 8-5. Block Erase



AT25DFX SERIAL FLASH DRIVER

- Existing driver within Atmel Studio for these flash ICs
- You can roll your own driver, or try using theirs
  - Theirs will probably be heavier code-wise, but have more protection / could get you moving more easily
  - http://asf.atmel.com/docs/3.32.0/samd21/html/asfdoc_common2_at25dfx_basic_use.html
- SPI Driver Documentation (for rolling your own)
  - http://asf.atmel.com/docs/3.32.0/samd21/html/asfdoc_sam0_sercom_spi_exqsg.html

APPENDIX


Generic MCU Boot-loader
Branch
Enter Bootloader?
N
Y
Application
Run Application
Enter Bootloader Request?
Y
N
Reset
Soft Reset
Bootloader
Initialize Bootloader
Command Received?
Y
N
Execute Command
Exit To Application?
Y
N

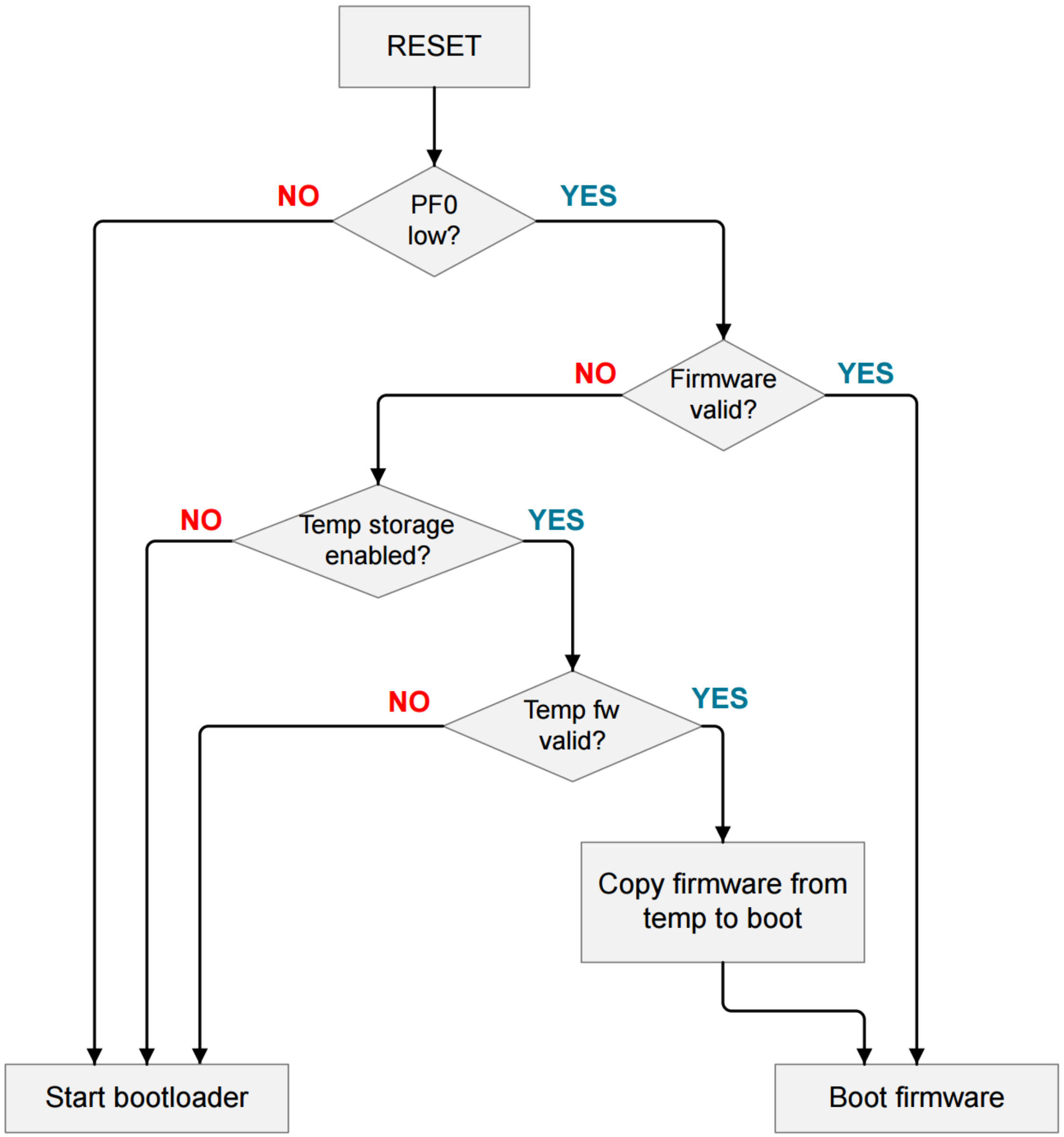
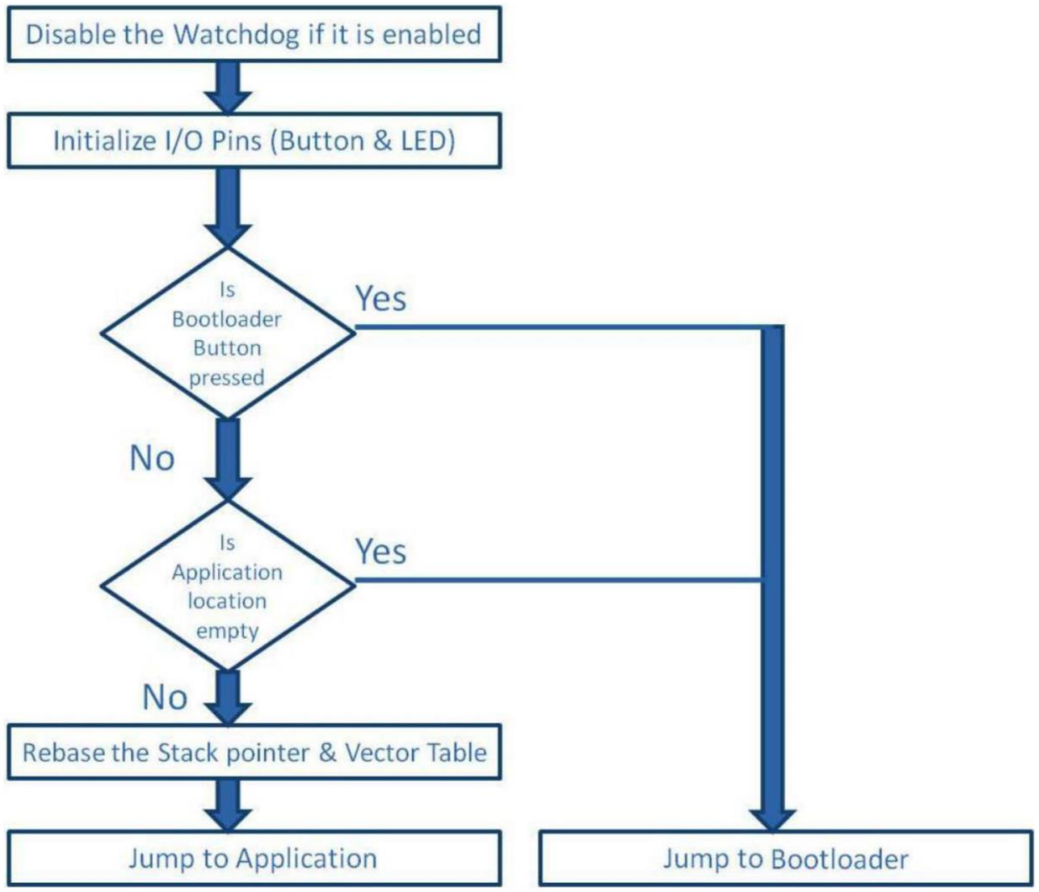


Figure 3.1. Bootloader State Machine

- In this case, PF0 is a hardware pin checked at boot – the bootloader is not entered unless that pin is low.
  - This is entering a boot mode by holding down a button.
- Note the firmware validity check for the **existing firmware in MCU memory**.
  - If there isn't a valid image, the device remains in bootloader mode.
- Temp storage for us is the firmware image stored in external flash memory
- Learn more:
  - https://www.silabs.com/documents/public/application-notes/an0060-bootloader-with-aes-encryption.pdf

ATMEL BOOTLOADER REFERENCES

Figure 4-3. Boot Condition Check Flowchart



- In this case, PF0 is a hardware pin checked at boot – the bootloader is not entered unless that pin is low.
- USB / UART bootloader for SAM D21:
  - http://www.atmel.com/Images/Atmel-42366-SAM-BA-Bootloader-for-SAM-D21_ApplicationNote_AT07175.pdf
- USB MSC: Page 31-32 has some good references
  - http://www.atmel.com/images/atmel-42352-sam-d21-xpro-usb-host-msc-bootloader_training-manual_an8185.pdf